
Epiphany

OPERATION CATALOG

A companion to the Core Specification

Version 0.9.0 — Space-relative CMN transposition

Normative for the operation kinds it defines

Contents

1	About This Companion	2
1.1	Relationship to the Core Specification	2
1.2	Conformance Profiles	3
2	The Catalog Framework	4
2.1	Value-Typed Payloads	4
2.2	Per-Primitive Schema Template	4
2.3	Reduction-Discipline Coverage	5
3	Ko — Representative Primitives	6
3.1	InsertEvent	6
3.2	DeleteEvent	7
3.3	RespellPitch	7
3.4	ModifyEvent	8
3.5	Identified-Pitch Operations	9
3.6	Transpose (frozen; replay only)	9
3.7	TransposeInterval	11
3.8	CreateCrossCutting	13
3.9	DeleteCrossCutting	14
3.10	ModifyCrossCutting	14
3.11	ChangeRegionTimeModel	15
3.12	Structural Containers	15
3.13	CreateStaff	16
3.14	Repeat Structures	17
3.15	SetUserSystemBreak	18
3.16	Score Settings	18
3.17	Meter and Tempo Overwrites	19
3.18	SetStaffLayout	20
3.19	DeclareTransaction	20
3.20	ResolveConflict (meta-operation)	20
3.21	ResolveEquivocation (meta-operation)	21
3.22	UndoTransaction (meta-operation)	22
4	v0 → v1 Payload Migration	24
5	K1 — Framework Slots (Phase 3)	25

About This Companion

The *Operation Catalog* is a companion to the Epiphany Core Specification. It fulfils the open question the core specification raises in its *Operation Catalog Conformance* section (Chapter 6, SEMANTIC OPERATIONS AND CONCURRENT REDUCTION, the `sec:semops:catalog` open question), which states that the catalog “is normative once published; this specification is non-final until the catalog is delivered.” This release delivers the catalog *framework*, the **Ko representative primitive set**, and the **M2 broad-Ko groups** (the event/pitch leaf-field, cross-cutting CRUD, structural-container, and score-settings operations the Phase 2 slice exercises) — the operation kinds the Phase 2 visible slice and binary format actually exercise, each fully specified in Chapter 3. The remaining items of the full 60–80-primitive catalog are drafted as framework slots (Chapter 5) and completed in Phase 3.

1.1 Relationship to the Core Specification

This companion does not restate the operation framework; it *references* it. The framework — operation identity and stamps, the hybrid-logical-clock monotonicity rule, the dotted-version-vector causal context, the order-independent operation-slot model and equivocation, the canonical reduction order, the four-phase lifecycle, conflict records and the conflict registry, re-anchoring, transactions, forward undo, and the LWW discipline — is the core specification’s Chapter 6. This companion defines, for each operation kind, the *payload schema* and how that kind *instantiates* the framework’s reduction, conflict, undo, and re-anchoring rules.

It also consumes, rather than re-deriving, the **ratified byte-convention baseline** (core specification Chapter 8, BINARY FORMAT COMPANION, requirement `req:format:codec-conventions`, and the CANONICAL BYTE-LAYOUT REFERENCE appendix): little-endian integers, a single discriminant byte per tagged union, `u32` length prefixes on every variable-width leaf, and raw UTF-8 free text. Operation-payload encodings are expressed in terms of that baseline; the literal wire layout of each payload is the Binary Format companion’s (Agent J’s) to pin, in coordination with this catalog.

RATIONALE

The catalog is versioned *independently* of the core specification (independent semver). Operation kinds are added over time; the catalog should evolve — adding primitives, refining conflict cases — without forcing a core specification revision. The core specification changes only when the *framework* changes.

1.2 Conformance Profiles

A **Phase-2 profile** implementation **MUST** implement every primitive in Chapter 3 — the representative set and the M2 broad-Ko groups — with the schema, reduction rule, conflict cases, undo semantics, and re-anchoring behaviour defined there. The former create-score/create-canvas framework slots of Chapter 5 are *retired* (Pass 12): they are permanently outside the operation set, so no operation kind exists to reject.

Version 0.6.0 (Pass-12 G-ratification). No payload byte changes. Ratified: ResolveConflict beyond the concurrent case (P12-K4), ResolveEquivocation edge semantics (P12-K6) and the profile-policy deferral (P12-K5), the RespellPitch migration fallback as long-term (P12-K1), undo strand-block conflict-kind reuse (P12-K10) and the idempotence asymmetry (P12-K11), the Transpose prototype pin (P12-K2 deferral), the cross-region slur AND advisory (P12-K12), and the create-score/canvas retirement (P12-K8). Appended vocabulary (minor, append-only): SystemDerivedContentImmutable (12, P12-K3) and RecreateContentMismatch (13, P12-K9) in PreconditionFailureReason.

Version 0.8.0 (Push 4a, transpose algebra). Adds one operation kind, TransposeInterval (Section 3.7), carrying the existing TranspositionInterval of core Chapter 2. It transposes scale position, refuses atomically rather than saturating, takes a genuine *set* of targets, and propagates the resulting spelling. Transpose (Section 3.6) is **frozen**: its saturation, its nominal-blindness, and its duplicate-sensitivity are pinned as normative replay semantics and no longer authored. This supersedes the 0.6.0 pin of the Transpose prototype (P12-K2), which anticipated a payload schema-major landing with the tuning catalog; neither proved necessary. Appended vocabulary (minor, append-only): AcousticRealizationPinned (14) and TranspositionOutOfRange (15) in PreconditionFailureReason; PitchSpaceMismatch (6) is un-reserved and now produced for a non-Cmn position. That case reads the position discriminant and does not require a pitch-space registry.

Version 0.9.0 (P13-S2, space-relative CMN). Broadens the PitchSpaceMismatch (6) case for TransposeInterval: a Cmn target whose enclosing chromatic structure or nominal mapping cannot be established now refuses under core requirement req:pitch:space-capability-refusal, rather than receiving guessed 12-chromatic arithmetic. This capability check is distinct from the 0.8.0 non-Cmn discriminant check and is replaced by structural registry resolution in Push 4b. No payload bytes change and no PreconditionFailureReason is appended; assignments 10 through 15 remain exactly as ratified.

The Catalog Framework

2.1 Value-Typed Payloads

Every operation payload in this catalog is **value-typed**: it carries the real graph values its effect introduces, not identifiers that point at values in some ambient graph. An `InsertEvent` carries the whole `Event`; a `RespellPitch` carries the whole `PitchSpelling`; a `CreateCrossCutting` carries the whole tie, slur, beam, or spanner.

RATIONALE

The vo prototype carried *identifier-only projections* — an `InsertEvent` held an `EventId` plus reduction-relevant scalars; a `RespellPitch` held a content-hash *fingerprint* of the new spelling. That was sufficient to make an envelope hashable and to drive reduction, but it is **not durable**: an operation replayed in a fresh context — a backup restore, a cross-tool round-trip — needs the full value, which an identifier-only payload cannot supply. Value-typed payloads are what make an Epiphany document portable.

REQUIREMENT 2.1

A value-typed payload field **MUST** be encoded by emitting the field's value under the core specification's canonical value encoding (`req:format:codec-conventions`), framed by a `u32` little-endian length prefix. Decoding **MUST** be the exact inverse and **MUST** reject trailing bytes within the framed region. The encoding introduces no byte layout beyond the ratified baseline: a value's bytes here are byte-for-byte the bytes the whole document codec emits for that value.

2.2 Per-Primitive Schema Template

Each catalog primitive is specified under a fixed six-part template. A new primitive is a schema-fill against this template, not a fresh design:

Payload schema The value-typed fields the operation carries, with their core-specification types.

Canonical encoding The field order and framing, consuming Requirement 2.1. (The literal byte layout is the Binary Format companion's; this catalog fixes the *field set and order*.)

Reduction rule How the operation mutates canonical state when it is reached in canonical reduction order — the preconditions it checks, the objects it mints or tombstones, and the bookkeeping it records.

Conflict cases The conflict records the operation can produce, by `ConflictKind`, and which participant materialises.

Undo semantics The compensating effect of undoing a transaction that contains the operation, under each `UndoPolicy` (`StrictInverse` / `BestEffort` / `Cascade`).

Re-anchoring The behaviour when an object the operation references is tombstoned before or concurrently with it.

2.3 Reduction-Discipline Coverage

The Ko representative set is chosen so that, between them, the primitives exercise *every* reduction discipline the framework defines: position-keyed insert with system-voice promotion; delete-wins with tombstones, tuple compensation, and cross-cutting re-anchoring; field-overwrite with last-writer-wins and structural-field-collision conflicts; set-union creation; structural time-model migration; an LWW advisory; atomic transactions with descriptor precedence; and the two meta-operations (conflict resolution and forward undo). A primitive added later that reuses one of these disciplines inherits its reduction, conflict, undo, and re-anchoring treatment.

Ko — Representative Primitives

This chapter is normative under the Phase-2 profile. Each primitive’s reduction, conflict, undo, and re-anchoring behaviour is the behaviour the core specification’s Chapter 6 defines for its discipline; the description here states how the primitive instantiates it. The reference implementation is `epiphany-ops` (the `payload`, `reduce`, and `migrate` modules).

3.1 InsertEvent

Payload schema. `InsertEventOp { staff_instance: StaffInstanceId, event: Event }`. The voice, region-local position, duration, and pitch identities are read from the `Event` value; the `staff_instance` is retained alongside it so the system-voice promotion derivation is total without a containment walk.

Canonical encoding. `staff_instance`, then the length-framed canonical bytes of `event`.

Reduction rule. A position-keyed insert. Preconditions: the event’s duration is positive; the event id is neither live nor tombstoned; in graph-aware reduction the target voice exists in a metric region and the event’s pitch ids are fresh. The event and its pitches are minted live; the voice is created on first use. Concurrent inserts whose half-open duration intervals overlap in the same voice are resolved by an order-independent promotion pre-pass: the lower-`OperationId` insert is retained in the original voice, and each overlapping loser is promoted to the deterministic system voice `derive_promoted_voice_id(staff_instance, voice, winner, loser)` and tagged `VoicePromoted`.

Conflict cases. None at reduction time; promotion is a deterministic repair, not a conflict.

Undo semantics. Undoing the enclosing transaction tombstones the minted event, its pitches, and any promoted voice. `StrictInverse` conflicts if any was already tombstoned or modified; `BestEffort` tombstones the survivors; `Cascade` is `StrictInverse` over the same minted set (dependent-closure undo is a Phase-3 refinement).

Re-anchoring. Not applicable (the operation mints, it does not reference a pre-existing object that could be tombstoned).

3.2 DeleteEvent

Payload schema. `DeleteEventOp { event: EventId, tuplelet_compensation: TupleletCompensation }`, where `TupleletCompensation` is `NotInTuplelet`, `ReplaceWithRest { rest: Rest }` (value-typed replacement `rest`), `RewriteTuplelets { tuplelets }`, or `CascadeDeleteTuplelets { tuplelets }`.

Canonical encoding. `event`, then the tuplelet-compensation discriminant and its payload (a length-framed `Rest` value for `ReplaceWithRest`).

Reduction rule. Delete-wins: the event and its contained pitches are tombstoned (retaining their identifiers). Tuplelet compensation, when present, adds the replacement `rest` live or tombstones the cascaded tuplelet group. Concurrent deletes of the same event are idempotent.

Conflict cases. None for the delete itself. Graph-aware reduction refuses an ill-formed tuplelet compensation as a precondition failure (no-op).

Undo semantics. An insert-shaped compensation re-introduces the tombstoned content; for the prototype’s minted-object model this is the inverse of the tombstone set, with the policy treatment described under `InsertEvent`.

Re-anchoring. Tombstoning the event runs the framework’s re-anchoring rule table over every cross-cutting structure that referenced it: a tie cascade-deletes; a comment or analytical annotation orphans (user content is never silently deleted); a beam truncates while ≥ 2 members survive and otherwise cascade-deletes; a slur or spanner re-anchors to the nearest surviving endpoint while ≥ 1 survives and otherwise cascade-deletes.

3.3 RespellPitch

Payload schema. `RespellPitchOp { pitch: PitchId, spelling: PitchSpelling }` — the full spelling value (`v1`), not a fingerprint.

Canonical encoding. `pitch`, then the length-framed canonical bytes of `spelling`.

Reduction rule. A last-writer-wins field overwrite, keyed by `pitch`. Precondition: the pitch is live. The resolved spelling is the one carried by the operation latest in canonical order. Two respellings of one pitch that are causally ordered overwrite intentionally; two *concurrent* respellings with *equal* spelling reduce idempotently.

Conflict cases. Two concurrent respellings of one pitch with *differing* spellings produce a `StructuralFieldCollision` conflict recording the winner (later in canonical order), the loser, and the field `spelling`. The winner materialises and carries the `Conflicted` effect tag.

Undo semantics. Value-restoring (Section 3.22): undo restores the pre-operation spelling (or removes the spelling if the operation introduced the first one), under the active policy.

Re-anchoring. If the target pitch is tombstoned, the respelling is a no-op (`TargetTombstoned`).

Migration (ratified Pass 12, closing P12-K1). A `v0 RespellPitch` carried only a content-hash *fingerprint* of the spelling. The fingerprint cannot be inverted to a `PitchSpelling` without a side table, so the `v0→v1` migration (Chapter 4) recovers the spelling from the score graph context — an explicit per-pitch spelling attachment whose canonical bytes hash to the fingerprint —

and, when the context lacks it, declares the envelope unmigratable (the bundle opens read-only). This read-only fallback is the *long-term* disposition: no richer vo corpus is or will be required (no production vo corpus exists). It remains the one representative payload that is not self-contained under migration.

3.4 ModifyEvent

Payload schema. `ModifyEventOp { event: Event }` — the full replacement `Event` value (`v1`). The identity (and therefore the LWW key) is read from the value.

Canonical encoding. The length-framed canonical bytes of `event`.

Reduction rule. A last-writer-wins field overwrite keyed by event id. Precondition: the event is live. The resolved value is the one carried by the operation latest in canonical order; two causally ordered modifications overwrite intentionally, and two *concurrent* modifications with *equal* value reduce idempotently. Graph-aware reduction overwrites the event in place, and a modification that *moves* a metric event (a different region-local `Musical` position or duration) is *materialised*: the owning voice is re-sorted by ascending position (id-tiebroken — the same order an insert maintains), preserving `VoiceEventsSortedNonOverlap` (Chapter 5 invariant 3). To keep that invariant, a placement change carries a *placement precondition*, read from the reducer’s canonical voice-occupancy index (graph-independent, so graph-free and graph-aware reduction agree): a move with a non-positive span, or one that would overlap another live event in the voice, is refused as a clean precondition no-op (`EventDurationInvalid`) rather than skipped silently. A materialised move updates the occupancy index, so a later insert sees the freed or changed span. A placement change of a *non-metric* event is recorded in the bookkeeping but not applied to the graph (re-sorting a non-metric voice is a deferred refinement); a malformed (empty pitched) replacement is likewise recorded but not materialised; same-placement field edits apply in place, preserving voice membership. Partial trimming of a tuple member remains a later refinement.

System-derived content immutability (ratified *Pass 12*, closing *P12-K3*). A modification that would rewrite the *intrinsic content* of a pitch whose identifier lives in the `SYSTEM_DERIVED` namespace is refused as a clean precondition no-op with the appended reason `SystemDerivedContentImmutable` (discriminant 12): the identifier is content-derived, and an in-place rewrite would silently invalidate its derivation (core specification, Chapter 5 system-derived identity). The same precondition applies to `ModifyIdentifiedPitch` (Section 3.5). The sanctioned path is minting a replacement pitch.

Conflict cases. Two concurrent modifications of one event with *differing* values produce a `StructuralFieldCollision` on the field `event`, recording the winner (later in canonical order) and the loser.

Undo semantics. Value-restoring (Section 3.22): undoing the enclosing transaction restores the event’s chain-predecessor value, conflicting (`StrictInverse`) or skipping (`BestEffort`) when a later modification has superseded it.

Re-anchoring. If the target event is tombstoned, the modification is a no-op (`TargetTombstoned/TargetMissing`).

3.5 Identified-Pitch Operations

Payload schema. `InsertIdentifiedPitchOp { event: EventId, pitch: IdentifiedPitch }` mints a pitch into a live event; `DeleteIdentifiedPitchOp { pitch: PitchId }` tombstones one; `ModifyIdentifiedPitchOp { pitch: PitchId, value: Pitch }` overwrites a pitch’s acoustic / scale-position value (distinct from `RespellPitch`, which overwrites only the *spelling*).

Canonical encoding. Insert: event, then the length-framed pitch value. Delete: pitch. Modify: pitch, then the length-framed value.

Reduction rule. The pitch-level analogues of the event-level mint, delete, and field overwrite, inheriting their disciplines (Sections 3.1, 3.2, and this chapter’s field-overwrite treatment). **A note and a rest are the same slot under pitch add/remove** (normative): deleting the *only* pitch of a single-pitch note degrades the event to a Rest of the same id/voice/position/duration rather than leaving an empty pitched event (Chapter 5 forbids the empty chord, `ArenaError::EmptyPitchedEvent`); inserting a pitch into a rest is the dual, promoting it to a one-pitch note. This preserves the delete-wins / mint disciplines and keeps the graph consistent with the bookkeeping, which tombstones or mints the pitch object either way.

Conflict cases. Insert and delete: none (mint is set-union; delete-wins is idempotent). Modify: two concurrent differing writes of one pitch produce a `StructuralFieldCollision` on the field `pitch`.

Undo semantics. Insert: undoing the enclosing transaction tombstones the minted pitch, re-resting the event if it was the only one (Section 3.22). Modify: value-restoring (Section 3.22) — the pitch’s chain-predecessor value is restored unless superseded. Delete: still not inverted (re-introducing a tombstoned pitch is the deferred resurrection case, P11-C8).

Re-anchoring. An operation whose target event or pitch is tombstoned is a no-op; tombstoning a pitch runs the cross-cutting re-anchoring table over any structure that referenced it (see `DeleteEvent`).

3.6 Transpose (frozen; replay only)

REQUIREMENT 3.1

Transpose is **frozen**. Its reduction rule is fixed at the semantics described in this section, including the saturation and the duplicate-sensitivity, for as long as any stored operation carries it. A conforming implementation **MUST** reduce it exactly as specified here and **MUST NOT** emit it from new authoring; new transpositions are authored as `TransposeInterval` (Section 3.7).

Payload schema. `TransposeOp { targets: Vec<PitchId>, chromatic_steps: i32 }`. Pitch identifiers are preserved. The operation rewrites *scale position* (the `Cmn` alteration), not acoustic content.

Canonical encoding. targets in ascending canonical-byte order, then `chromatic_steps` as a little-endian `i32`. The sequence is *sorted but not deduplicated*: it is a multiset on the wire.

Reduction rule. An order-dependent content overwrite. For each live, non-system target *occurrence* in `targets`, the target’s `Cmn` alteration is shifted by `chromatic_steps` and clamped to the `i8` bound. Consequences, all normative for replay:

- A target appearing k times is shifted k times.
- The nominal and octave never change: transposing C4 by +12 yields C4 with `alteration` = 12, not C5.
- A shift past the `i8` bound saturates silently, and the operation still reports `Applied`. Transposition under this rule is therefore not invertible.
- A target whose position is not `Cmn` is left unchanged, and the operation still reports `Applied`.

RATIONALE

Every bullet above is a defect (P12-K2), and none of them may be repaired in place. An operation is history: a replica replaying a stored `Transpose` must reconstruct the state its author saw, and a “corrected” reduction rule would silently rewrite every score that ever used one. The defects are therefore *pinned*, not fixed, and the repair ships as a new operation kind whose discriminant no historical operation carries. Ratified Push 4a; supersedes the Pass 12 pin, which anticipated a payload schema-major and a dependency on the tuning catalog. Neither proved necessary: transposition acts on scale position, and appending an operation kind is a schema *minor* (Binary Format companion, requirement `req:binfmt:kind-discriminants`).

Conflict cases. None — composition is deterministic in canonical order (a deterministic repair, not a conflict).

Undo semantics. `Transpose` mints nothing, so the prototype’s minted-object `undo` (Section 3.22) does not negate it, and it records no write into the pitch’s value chain, so *value-restoring undo does not restore it either*: undoing a transaction containing a `Transpose` leaves the shifted pitch shifted. An inverse-interval `undo` does not exist (under saturation the inverse is not a function) and remains a deferred refinement (P11-C8).

This too is frozen. Making `Transpose` record into the value chain would not change its own reduction rule, but it *would* change what a stored history `{Transpose, UndoTransaction}` replays to — from “the pitch stays shifted” to “the pitch returns” — which is a change in what an existing document means. `TransposeInterval` records its write and *is* undoable (Section 3.7); the frozen operation is permanently not. That asymmetry is another reason never to author it.

Re-anchoring. Tombstoned targets are skipped (the transpose applies only to live pitches). `SYSTEM_DERIVED`-namespace targets are likewise *skipped* (ratified Pass 12, P12-K3): their intrinsic content is immutable, and an in-place alteration shift would desynchronize the content from the `id`’s derivation inputs. A transpose whose live targets are *all* system-derived reduces as a precondition no-op (`SystemDerivedContentImmutable`).

3.7 TransposeInterval

Payload schema. `TransposeIntervalOp { targets: CanonicalSet<PitchId>, interval: TranspositionInterval }`, reusing the diatonic/chromatic pair the core specification defines in Chapter 2, TRANSPOSITION AND THE INTERVAL TYPE — the same value an `Instrument` carries as its written-versus-sounding interval. Pitch identifiers are preserved; the operation rewrites scale position and leaves acoustic realization untouched.

REQUIREMENT 3.2

`targets` is a **set**. It **MUST** be encoded as a sequence in ascending canonical-byte order with *strictly increasing* elements; a decoder **MUST** reject a sequence containing a duplicate rather than normalize it. A transposition is a function of *which* pitches it names, never of how many times it names them.

Canonical encoding. The strictly-increasing `targets` sequence, then `interval.diatonic_steps` and `interval.chromatic_steps`, each a little-endian `i32`.

Reduction rule. An order-dependent content overwrite. Tombstoned and `SYSTEM_DERIVED` targets are skipped, exactly as for `Transpose`; if no mutable target remains the operation reduces as the corresponding precondition no-op. Otherwise, every remaining target **MUST** be transposable under core specification requirement `req:pitch:transposition`, and each is transposed by `interval`.

REQUIREMENT 3.3

If any mutable target is *not* transposable, the operation **MUST** reduce to a no-op that changes no pitch, reporting the precondition failure of the first such target in canonical order. It **MUST NOT** transpose the remaining targets. The four cases and their `PreconditionFailureReason` discriminants:

- a non-Cmn `scale_position.position` $\Rightarrow$ `PitchSpaceMismatch (6)`;
- a Cmn position whose enclosing chromatic structure or nominal mapping cannot be established $\Rightarrow$ `PitchSpaceMismatch (6)`;
- an `AcousticRealization::AbsoluteHz` realization $\Rightarrow$ `AcousticRealizationPinned (14)`;
- an `alteration'` or `octave'` that does not fit its field $\Rightarrow$ `TranspositionOutOfRange (15)`.

REQUIREMENT 3.4

A `TransposeInterval` **MUST** record the spelling its interval determined as a `SpellingAttachment` with source `SpellingSource::Propagated`, per the core specification's Chapter 2, SPELLING SOURCES. The attachment's `from` is the transposed pitch's own identifier, which the operation preserves.

That attachment alone is *not sufficient*, because the default precedence ranks `UserChosen` and `Imported` above `Propagated` (Chapter 2, CONFIGURABLE PRECEDENCE). A

`TransposeInterval` **MUST** therefore also **move** every engraved-layer, pitch-scoped, explicit spelling attachment on each target that is not itself `Propagated`, preserving each attachment’s `source`, `priority`, and `layer`. A spelling is moved by transposing its *nominal* diatonically and taking whatever accidental the transposed pitch then requires at that staff position. If any such attachment cannot be written at the transposed position, the operation **MUST** refuse, atomically, with `TranspositionOutOfRange`.

RATIONALE

Two different failures hide here, and the propagated attachment addresses only the first. For a pitch with *no* authored spelling, the pre-pass would re-infer, and may pick the enharmonic the interval did not choose. The propagated attachment pins the interval’s determination.

For a pitch *with* an authored spelling, the propagated attachment is outranked and changes nothing: a C₄ the author spelled “C”, sharpened to C-sharp 4, would still resolve to `Authorled(UserChosen)` and engrave a C natural — the accidental simply disappears, and the notehead names a pitch that is not there. So the authored spelling is moved instead of being left or discarded. It is moved by its **nominal**, because the nominal is what carries the author’s enharmonic decision: an author who wrote B-sharp 3 rather than C₄ chose the letter B, and a perfect fifth up must give F-double-sharp 4, not G. Discarding the attachment would silently lose that decision; re-inferring it from the transposed pitch would give G.

A transposed `UserChosen` spelling is still the user’s choice, so its source is preserved. `Imported` likewise: an imported attachment asserts “this pitch is written thus”, and once the pitch moves that assertion must move with it or become false. Import fidelity is a property of the source file, which a transposition does not touch.

Conflict cases. None — composition is deterministic in canonical order.

Undo semantics. Nothing is minted. `TransposeInterval` **MUST** record each transposed pitch’s new value into that pitch’s value chain, and that pitch’s resulting *engraved spelling set* into the spelling-set chain, so that value-restoring undo recovers the pre-transpose pitch *and* its pre-transpose spelling. Restoring the pitch alone would leave a notehead spelled for a pitch that no longer exists. This is where `TransposeInterval` departs from the frozen `Transpose`, which records nothing and is therefore not undoable.

REQUIREMENT 3.5

The pitch’s engraved spelling set is a single undo key with *more than one writer*. Every operation that changes it — `RespellPitch` and `TransposeInterval` alike — **MUST** record a write on that key. Consequently a `RespellPitch` causally prior to a transposed transaction is that transaction’s chain-predecessor and is restored by its undo; and a `RespellPitch` canonically *later* supersedes the transaction’s write, so a `StrictInverse` undo conflicts rather than erasing it, per the general superseded-writer rule.

A pitch’s value and its engraved spelling set **MUST** undo as one unit: if either is superseded, neither is restored. A `BestEffort` undo **MUST NOT** restore a pitch’s earlier value

while leaving a spelling authored against the value it has since taken. This unit is keyed on the *pitch*, and on which keys the *transaction* wrote — not on which operation wrote them, and not on `TransposeInterval` in particular. A transaction whose members write the two halves separately is coupled the same way: an editor’s “move note” is a `ModifyIdentifiedPitch` (the value) together with a `RespellPitch` (the spelling set), and a later respell makes a `BestEffort` undo skip both. That breadth is intended. A transaction that wrote only one of the two keys is unaffected, since an unwritten key yields no supersession.

RATIONALE

Ratified as P13-S3. An implementation may keep the spelling-set chain physically separate from whatever chain drives `RespellPitch`’s last-writer-wins conflict detection — the reference implementation does, because folding transposes into that chain would make a concurrent respell *conflict* with a transpose and would move the canonical bytes of every existing history. What it may not do is let one writer own the key. A chain with a single writer cannot see the other’s prior value (so undo erases it) and cannot see the other’s later value (so undo overwrites it).

An inverse interval usually exists $((-d, -c))$, because the reduction never saturates — but not always: `i32::MIN` has no negation. Undo does not rely on it.

Re-anchoring. As `Transpose`.

3.8 CreateCrossCutting

Payload schema. `CreateCrossCuttingOp { structure: CrossCuttingValue }, where CrossCuttingValue` is the typed value of a `Tie`, `Slur`, `Beam`, or `Spanner`. Its identity and referenced endpoints are read from the value.

Canonical encoding. A discriminant for the structure kind, then the length-framed canonical bytes of the structure value.

Reduction rule. Set-union creation: the structure is minted live if its id is not already live and every referenced endpoint is live; a second create of a live id is idempotent. Graph-aware reduction materialises the structure with its full fields.

Conflict cases. None (all-or-nothing creation; union is deterministic).

Undo semantics. Undo tombstones the minted structure, under the active policy.

Re-anchoring. The structure participates in the re-anchoring rule table when one of its endpoints is later tombstoned (see `DeleteEvent`).

Authoring advisory (ratified Pass 12, closing P12-K12). The slur-spanning advisory reads `Region.permits_spanning_slurs` *conjunctively*: a slur (or other spanner) whose endpoints lie in different regions passes the advisory only when *both* endpoint regions permit spanning. The check is authoring-time only and never alters reduction.

Migration coverage. The $v_0 \rightarrow v_1$ migration (Chapter 4) reconstructs the event-anchored `Tie`, `Slur`, and `Beam` from the v_0 reference (id plus event endpoints). A `Spanner` is anchored by `TimeAnchors` rather than a fixed pair of event endpoints, so its full value is not reconstructable from the v_0 event-reference projection; a `Spanner-create` is therefore reported unmigratable (read-only), alongside the respell case of `P12-K1`, and remains so. A faithful spanner migration awaits a richer v_0 projection that carries the anchors — a Phase-3 / Pass-12 extension, not yet implemented.

3.9 DeleteCrossCutting

Payload schema. `DeleteCrossCuttingOp { structure: TypedObjectId }` — the structure named by the same key the set-union creation and the re-anchoring table use; it **MUST** be a cross-cutting kind (`Tie/Slur/Beam/Spanner`).

Canonical encoding. The canonical bytes of `structure`.

Reduction rule. Delete-wins: the structure is tombstoned (its identifier retained). A second delete of the same structure is idempotent; deleting a missing or non-cross-cutting id is a no-op precondition failure. Graph-aware reduction removes the structure from the score.

Conflict cases. None (delete-wins is idempotent).

Undo semantics. A delete mints nothing, so the prototype’s minted-object undo (Section 3.22) does not re-introduce the tombstoned structure (`P11-C8`).

Re-anchoring. The deletion is direct (the structure is the target, not a referenced endpoint); it does not itself trigger the endpoint re-anchoring table.

3.10 ModifyCrossCutting

Payload schema. `ModifyCrossCuttingOp { structure: CrossCuttingValue }` — the full replacement value (v_1). The replacement keeps the structure’s identity but may change its endpoints and per-kind fields; the LWW key is the structure’s `CrossCuttingValue::id`.

Canonical encoding. The discriminant and length-framed bytes of the `CrossCuttingValue` (as for `CreateCrossCutting`).

Reduction rule. A last-writer-wins field overwrite keyed by structure id. Precondition: the structure is live. The reduction re-derives the structure’s endpoints from the new value, so a later re-anchoring sees them. A malformed replacement is a precondition no-op — in particular a beam whose membership falls below the two-member minimum is refused rather than materialised. Graph-aware reduction overwrites the structure in place.

Conflict cases. Two concurrent differing modifications of one structure produce a `StructuralFieldCollision` on the field `cross_cutting`.

Undo semantics. Value-restoring (Section 3.22): undo restores the structure’s chain-predecessor value unless superseded.

Re-anchoring. If the target structure is tombstoned, the modification is a no-op; re-deriving

the endpoints lets a subsequent endpoint tombstone re-anchor the structure through the standard table (see `DeleteEvent`).

3.11 `ChangeRegionTimeModel`

Payload schema. `ChangeRegionTimeModelOp { region: RegionId, new_time_model: RegionTimeModel, declared_incompatible: Vec<EventId>, remapping: PositionRemapping }` — the full target model value (`v1`).

Canonical encoding. `region`, the length-framed `new_time_model` value, the canonically-ordered `declared_incompatible` set, then `remapping`.

Reduction rule. Structural migration. The region adopts the target time model. Graph-aware reduction derives coordinate-kind incompatibilities from the region’s events (and from the remapping coverage) and refuses a migration that would violate the coordinate discipline.

Conflict cases. Concurrent same-region migrations produce a `StructuralFieldCollision` on the field `time_model`; a migration with incompatible events produces a `TimeModelMigrationFailure` naming the region and the incompatible events. A causally-later migration is re-evaluated against the first migration’s graph rather than conflicting.

Undo semantics. Undo restores the region’s prior time model under the active policy.

Re-anchoring. Not applicable.

OPEN QUESTION

P11-C6. The rich migration payload — a coordinate converter rather than a `declared_incompatible` list plus a `PositionRemapping` — remains the catalog’s to design when graph-aware migration is the only reduction path.

3.12 `Structural Containers`

Payload schema. Three create/delete pairs over the region hierarchy: `CreateRegionOp { region: Region } / DeleteRegionOp { region: RegionId }; CreateStaffInstanceOp { region: RegionId, instance: StaffInstance } / DeleteStaffInstanceOp { staff_instance: StaffInstanceId }; CreateVoiceOp { staff_instance: StaffInstanceId, voice: Voice } / DeleteVoiceOp { voice: VoiceId }`. Each create carries the full container value (`v1`); the reduction preconditions it bears *no typed child object* — an empty container.

Canonical encoding. Create: the parent id (where the schema names one), then the length-framed canonical bytes of the container value. Delete: the container id.

Reduction rule. Set-union creation of an *empty* container, and an *empty-only* delete-wins tombstone. A create mints the container live if its id is fresh and (for staff instance and voice) its parent is live; it preconditions the carried value to bear no typed child object (a region: no staff instances, barline-alignment groups, or graphic objects; a staff instance: no voices or measures; a voice: no events), since those carry distinct `TypedObjectIds` the reducer mints separately — so contents are added by subsequent operations. A delete is a delete-wins tombstone, but a *precondition no-op* (`ContainerNotEmpty`) unless the container has no live children — the caller

deletes contents first. Graph-aware reduction adds or removes the container and maintains the region’s staff extent so `RegionExtents` stays satisfied.

Conflict cases. None at reduction time: creation is set-union (a repeat create is idempotent), and the empty-only delete is a deterministic precondition gate, not a conflict.

Undo semantics. Undo of a *create* tombstones the minted container (Section 3.22); `StrictInverse` conflicts if it was concurrently mutated, with the policy treatment as for `InsertEvent`. A *delete* mints nothing, so the prototype’s minted-object undo does not re-introduce it (P11-C8).

Re-anchoring. Not applicable (the containers are minted/tombstoned by id; the empty-only precondition means a delete never strands live children).

3.13 CreateStaff

Payload schema. `CreateStaffOp { staff: Staff }` — the full global-staff value (*v1*): identity, name, abbreviation, instrument reference, default staff-line configuration, and optional group membership.

Canonical encoding. The length-framed canonical bytes of `staff`.

Reduction rule. Set-union creation of a global `Staff` on the score root, completing the structural-container family (Section 3.12) upward: *staff instances* reference global staves, and until this primitive existed a resolvable staff could only be base-seeded. A create mints the staff live if its id is fresh; a repeat create carrying a byte-identical value reduces idempotently, and a create whose id is already live with a *differing* value is a precondition no-op with the appended reason `RecreateContentMismatch` (discriminant 13; ratified Pass 12, closing P12-K9 — the former `TargetMissing` reuse misnamed the situation: the target is not missing, its content disagrees). The same reason applies to the carried `TimeSignature` (Section 3.17), the other re-create site at which the reducer retains the carried value for comparison. The structural-container creates (Section 3.12) are plain set-union — any repeat create of a live id reads `AlreadyApplied` without value comparison, since the carried value is preconditioned empty of children. Graph-aware reduction additionally preconditions that the referenced instrument is live and, when `group` is present, that the staff group resolves — the mint must leave the graph satisfying the reference-resolution invariants.

With staves mintable, `CreateStaffInstance` (Section 3.12) preconditions that the instance’s referenced `Staff` is live (previously the reference was satisfiable only from the seeded base, so the check was vacuous).

Conflict cases. None at reduction time (set-union; the differing-value re-create is a precondition gate, not a conflict).

Undo semantics. Undo of a create tombstones the minted staff (Section 3.22); `StrictInverse` conflicts if a live staff instance references it (tombstoning it would strand the instance).

Re-anchoring. Not applicable (a staff mint references no tombstonable anchor; there is no `DeleteStaff` in this catalogue revision — an empty-only staff delete mirroring the container discipline is a later schema-fill).

3.14 Repeat Structures

Ratified with the schema-major-2 revision: the dedicated authoring pair for `RepeatStructure` (core specification Chapter 5 REPEAT STRUCTURES). Repeats live in the cross-cutting registry but are *not* `CrossCuttingValue` kinds on the wire — the cross-cutting operations admit only `Tie/Slur/Beam/Spanner` (Section 3.8) — so the pair is a first-class primitive mirroring the cross-cutting disciplines.

Payload schema. `CreateRepeatStructureOp { repeat: RepeatStructure }` — the full value (schema major 2): identity, the start/end anchors, the `RepeatKind`, and the volta list. `DeleteRepeatStructureOp { repeat: RepeatStructureId }`.

Canonical encoding. Create: the length-framed canonical bytes of `repeat` (the schema-major-2 layout). Delete: the bare identifier. The create’s payload embeds the v2 `RepeatStructure` layout unconditionally (`kind` and `volts` are not optional fields), so under minimal stamping a block carrying one stamps schema major 2 — the create is *born at v2*. The delete’s payload is an identifier — a major-0 layout — so minimal stamping gives its blocks major 0; the kind discriminant itself is an append-only schema-*minor* vocabulary event (the same append mechanism as the Phase-3 tranche; the stamp itself always follows minimal stamping over the payload, per the Binary Format companion SCHEMA MAJOR 2).

Reduction rule. Set-union creation and a delete-wins tombstone, mirroring the cross-cutting family. A create mints the repeat live if its id is fresh and *every* event-referencing anchor site resolves to a live event — `start/end`, the kind’s jump targets (`DaCapo.end_target`, `DalSegno.segno/end_target`) and each volta’s `start/end` — since the mint must leave the graph satisfying the reference-resolution invariants; a dead anchor is a precondition no-op (`TargetMissing`). A repeat create of a live id reads `AlreadyApplied` without value comparison (the cross-cutting discipline; the `RecreateContentMismatch` scope of Section 3.13 is unchanged). A delete tombstones the repeat (idempotent on re-delete, delete-wins); deleting a missing id is a no-op precondition failure. Graph-aware reduction adds or removes the structure from the cross-cutting registry.

Conflict cases. None at reduction time (set-union creation; delete-wins is idempotent).

Undo semantics. Undo of a create tombstones the minted repeat (Section 3.22). A delete mints nothing, so the prototype’s minted-object undo does not re-introduce the tombstoned structure (P11-C8).

Re-anchoring. The structure participates in the re-anchoring rule table when a referenced event is later tombstoned: re-anchor to the nearest surviving anchor, cascade-delete only when none survives — as for spanners, spanning *every* anchor site (`start/end`, jump targets, volta spans). Among multiple surviving candidates — a case slurs and spanners never present, since their sole other endpoint is the forced survivor — “nearest” is currently realized as the deterministic identifier-order minimum (the same tie-break the two-endpoint collapse already used); proximity-aware (four-key) selection over a repeat’s surviving sites is a deferred refinement, exactly as the spanner row defers per-kind proximity bounds. The rule row is the core specification’s (Chapter 6 re-anchoring rule table, ratified with this pair).

Authoring advisory. The volta well-formedness constraints of core Chapter 5 (endings non-empty, 1-based, *strictly* ascending) are *advisory*: surfaced at authoring time under interactive validation, never enforced under reduction.

3.15 SetUserSystemBreak

Payload schema. `SetUserSystemBreakOp { region: RegionId, anchor: TimeAnchor, present: bool }` — the full anchor value (*v1*).

Canonical encoding. `region`, the length-framed `anchor` value, then the boolean.

Reduction rule. A last-writer-wins advisory. The break preference is recorded for the region keyed by the anchor’s *resolved musical position*; graph-aware reduction adds or removes the anchor from the region’s user system-break list.

Conflict cases. None (LWW advisory).

Undo semantics. Undo restores the prior advisory value for the (`region`, `resolved-position`) key.

Re-anchoring. Not applicable in the prototype (the advisory is keyed by resolved position; a tombstoned anchor target degrades to the region origin).

3.16 Score Settings

Payload schema. Three score-level field overwrites: `SetMetadataOp { metadata: ScoreMetadata }` overwrites the score singleton; `SetMetricGridOp { region: RegionId, grid: Option<MetricGrid> }` overwrites (or clears) a region’s default metric grid; `SetUserPageBreakOp { region: RegionId, anchor: TimeAnchor, present: bool }` is the page-break sibling of `SetUserSystemBreak`.

Canonical encoding. Metadata: the length-framed `metadata` value. Metric grid: `region`, then an `Option` discriminant and (when present) the length-framed `grid` value. Page break: `region`, the length-framed `anchor` value, then the boolean.

Reduction rule. Three field overwrites differing only in discipline. *SetMetadata* is an **advisory** last-writer-wins: the latest write in canonical order silently wins and the operation always applies — no working state and no conflict (the same discipline as `SetUserSystemBreak`, on the score singleton). *SetMetricGrid* is a **structural** field overwrite keyed by `region`: precondition the region is live and staff-based (a `FreeGraphic` region has no metric-grid slot — the op is a no-op there), and reject a grid whose meter sequence names a time signature that is not live (the Chapter 5 invariant forbids installing such a grid). *SetUserPageBreak* is a canonical LWW advisory keyed by the anchor’s resolved musical position, with the same staff-based precondition. Graph-aware reduction overwrites the metadata singleton, sets the region’s default metric grid, or adds/removes the page-break anchor under its resolved-position key (so two anchors resolving to one position occupy a single slot).

Conflict cases. `SetMetadata` and `SetUserPageBreak`: none (advisory LWW). `SetMetricGrid`: two concurrent differing grids for one region produce a `StructuralFieldCollision` on the field `metric_grid`.

Undo semantics. All three are value-restoring field overwrites (Section 3.22): undo restores the prior metadata, grid, or break preference from the key’s write chain unless superseded.

Re-anchoring. The advisory breaks degrade as for `SetUserSystemBreak`; the metric grid and metadata are keyed by `region` / singleton and do not re-anchor (a deleted region’s settings are no-ops — `TargetMissing`).

3.17 Meter and Tempo Overwrites

Payload schema. The finer-grained metric-model overwrites beneath the whole-grid `SetMetricGrid` (Section 3.16): `SetTimeSignatureOp` { `region`: `RegionId`, `anchor`: `TimeAnchor`, `time_signature`: `Option<TimeSignature>` } sets, replaces, or (None) removes the single `MeterChange` at the anchor’s resolved musical position in the region’s default metric grid, carrying the full `TimeSignature` value (`v1`); `SetTempoSegmentOp` { `region`: `Option<RegionId>`, `start`: `TimeAnchor`, `segment`: `Option<TempoSegment>` } sets, replaces, or removes the single tempo segment starting at the resolved position, in the score-level tempo map (`region`: `None`) or the region’s local map (`Some`; a set on a region with no local map creates one).

Canonical encoding. Time signature: `region`, the length-framed `anchor`, then an `Option` discriminant and (when present) the length-framed `time_signature` value. Tempo segment: an `Option` discriminant and (when present) `region`, then the length-framed `start`, then an `Option` discriminant and (when present) the length-framed `segment`.

Reduction rule. Last-writer-wins structural overwrites keyed by (`region`, `resolved position`) (`time signature`) and (`scope`, `resolved start`) (`tempo segment`). A carried `TimeSignature` is minted set-union under the same discipline as `CreateStaff`: fresh id mints; byte-identical re-carry is idempotent; a differing value under a live id is a precondition no-op (`RecreateContentMismatch`, Section 3.13). The time-signature value’s beat-group sum is validated at construction and again at decode, so a malformed value never reaches reduction. A tempo-segment write preconditions that the *resulting* map is well-formed (segments ordered and non-overlapping; a non-constant shape carries its end data; the carried segment’s own start equals the operation’s start key) — a write that would malform the map is refused as a precondition no-op (`TempoMapMalformed`). Graph-aware reduction applies the meter change to `default_metric_grid.meter_sequence` and the segment to the scoped tempo map.

Conflict cases. Two concurrent differing writes of one key produce a `StructuralFieldCollision` on the field `meter_sequence` or `tempo_segments` respectively, with the standard winner/loser recording; identical concurrent writes reduce idempotently.

Undo semantics. Value-restoring per Section 3.22: undo restores the key’s chain-predecessor value (or its absence).

Re-anchoring. An event-anchored `anchor/start` whose event is later tombstoned degrades by the framework’s anchor rules; the overwrite keys on the *resolved* position, so the recorded change survives its anchor.

OPEN QUESTION

P12-C5. A mid-region meter change authored by `SetTimeSignature` reduces cleanly and materialises into the grid, but the notational-decomposition pre-pass currently honours only a region’s *first* governing meter (P12-H4’s single-meter simplification), so the derived notation ignores the change until multi-meter decomposition lands. The reduction-level semantics are pinned here; the derived-annotation gap is P12-H4’s.

3.18 SetStaffLayout

Payload schema. `SetStaffLayoutOp { staff_instance: StaffInstanceId, instrument_override: Option<InstrumentId>, staff_lines_override: Option<StaffLineConfiguration>, visible: bool }` — the non-break layout advisories with a graph home: the staff instance’s three inline advisory fields, overwritten as a unit.

Canonical encoding. `staff_instance`, an `Option` discriminant and (when present) `instrument_override`, an `Option` discriminant and (when present) the length-framed `staff_lines_override`, then the boolean.

Reduction rule. A last-writer-wins *advisory* overwrite keyed by `staff_instance`. Preconditions: the staff instance is live; a present `instrument_override` resolves to a live instrument under graph-aware reduction. The richer engraving-override vocabulary (stem direction, note-head shape, custom positions) has no durable graph home yet and remains projected layout state — extending this primitive to cover it is staged with the data-model expansion.

Conflict cases. None (LWW advisory).

Undo semantics. Value-restoring per Section 3.22.

Re-anchoring. If the staff instance is tombstoned, the overwrite is a no-op (`TargetTombstoned`).

3.19 DeclareTransaction

Payload schema. `TransactionDescriptor { id: TransactionId, label: String, category: Option<TransactionCategory> }`. Value-complete in vo and unchanged.

Reduction rule. Records the descriptor. Member primitives reference the transaction id and **MUST** causally depend on the descriptor; the members reduce atomically (all-or-nothing) in canonical order.

Conflict cases. A missing descriptor or a member that does not causally follow it produces a `TransactionConflict`; any member failure rolls back the whole transaction and all members read `NoOp{TransactionConflict}`.

Undo / re-anchoring. Transactions are the unit of undo (Section 3.22); re-anchoring is per member.

3.20 ResolveConflict (meta-operation)

Payload schema. `ResolveConflictPayload { target: ConflictId, action: ResolutionAction }`. Value-complete.

Reduction rule. Transitions the target conflict’s resolution state. An action of `Dismiss` reaches the `Dismissed` state; any other action reaches `Resolved`. Re-resolving with the same action is idempotent; two concurrent resolves with differing actions produce a meta-conflict.

Beyond the concurrent case (ratified Pass 12, closing P12-K4). The earliest-applied-resolve-governs rule is *universal*: a causally-later resolve with a differing action does not supersede the first — it reduces `AlreadyApplied` — and *any* resolve targeting a `Dismissed` conflict likewise reads

`AlreadyApplied`. Intentional re-resolution is deliberately outside the `v1` operation set; a future dedicated operation (a `ReopenConflict`-class primitive) is the sanctioned path if it is ever needed. The meta-conflict record names both resolver operation ids in `caused_by`; conflict records themselves have no `TypedObjectId` kind (deliberate — they are materialized state, not graph objects), so the contested conflict is identified by the `equivocation`-style field key, not an object reference.

RATIONALE

Pass 11 added `ResolutionAction::Dismiss` (item 2.5) precisely so the `Dismissed` state is reachable by an authored operation rather than merely representable. The catalog records that `Dismiss` is the action that selects it (resolving the `v0` ambiguity P11-C10).

3.21 ResolveEquivocation (meta-operation)

Payload schema. `ResolveEquivocationPayload` { `target`: `OperationId`, `chosen`: `EnvelopeHash` } — the equivocated slot and the candidate envelope (by canonical-bytes hash) that shall stand. Value-complete.

Canonical encoding. `target` (16 canonical bytes), then `chosen` (32 bytes), per the codec baseline.

Reduction rule. Order-independent promotion of an equivocated slot (core specification Chapter 6, EQUIVOCATION): when the operation set holds an `Equivocated` slot for `target` and `chosen` names one of its candidates, the slot reduces as if it had always been `Single` with the chosen envelope — the chosen candidate contributes to canonical reduction at its own canonical position, and operations that were pending on the equivocated id unblock. Among multiple resolves naming the same slot, the one earliest in canonical order governs; a later resolve naming the *same* candidate reduces idempotently (`AlreadyApplied`). The resolve operation must itself occupy a `Single` slot; an equivocated resolve is excluded from reduction like any other equivocated slot. A resolved slot records no `OperationSlotEquivocated` anomaly; the losing candidates remain in the diagnostic candidate store only.

Conflict cases. Two resolves of one slot naming *differing* candidates produce a `StructuralFieldCollision` meta-conflict on the field `equivocation_resolution`, recording the governing resolve (earlier in canonical order) as winner and the later as loser, with both operations in `caused_by` — the same discipline as `ResolveConflict` meta-conflicts. Preconditions: a resolve whose `target` is not an equivocated slot, or whose `chosen` is not among the slot's candidates, is a precondition no-op.

Edge semantics (ratified Pass 12, closing P12-K6). Promotion is *single-pass*, not fixpoint: a promoted candidate that is itself a `ResolveEquivocation` does not govern a further promotion in the same reduction. A resolve in a *quarantined* replica segment is excluded from reduction and never governs. A resolve held *pending* by its own causal gaps still governs promotion — the verdict is a pure function of the slot map (set-level), while the resolve's own effect stays pending. The invalid-target/invalid-chosen no-op reuses `TargetMissing`; a dedicated reason was considered and rejected (the appended-reason budget is spent where a distinct verdict changes caller behavior, which it does not here).

Undo semantics. Mints nothing; not inverted under the prototype’s minted-object undo (P11-C8).

Re-anchoring. Not applicable (the payload references an operation slot, not a graph object).

RATIONALE

The core specification names three resolution paths for an equivocated slot: transport-level reconciliation, this explicit operation, and a profile-declared deterministic selection policy. This entry pins the schema for the explicit-operation path, which the core specification previously named only in prose. The profile-policy path is *deferred with a named landing site* (Pass 12 disposition of P12-K5): v1 profiles declare *no* selection function, and the hook’s definition belongs to the Profile Conformance companion when it is written — the reducer deliberately carries no policy hook until a profile can declare one.

3.22 UndoTransaction (meta-operation)

Payload schema. `UndoTransactionPayload { target: TransactionId, policy: UndoPolicy }`, with `UndoPolicy` one of `StrictInverse`, `BestEffort`, `Cascade`. Value-complete.

Reduction rule. A forward compensating edit computed against the materialised state at the undo’s canonical position (never literal time travel). The compensation has two parts.

Minted-object tombstoning (as before): every object the target transaction minted is tombstoned. `StrictInverse` conflicts (`TombstonedTarget`) if any minted object was already tombstoned; `BestEffort` tombstones the survivors.

Value restoration (this revision): for every last-writer-wins overwrite the target transaction performed — event and identified-pitch modification, respelling, cross-cutting modification, metadata, metric grid, meter change, tempo segment, staff layout, and the user break advisories — the reducer maintains, per overwritten key, the *canonical-order write chain* of (writer, value) pairs. Undoing the transaction restores each written key to its chain-predecessor value (or its absence, where the transaction introduced the first value), *provided the transaction’s write is still the key’s last writer*. Because the chain is keyed by canonical order, the restored value is a pure function of the operation set: permutation-invariant by construction. When a causally-later or canonically-later write has superseded the key, `StrictInverse` refuses the whole undo with a `TransactionConflict` conflict naming the undo and the superseding writer; `BestEffort` restores the still-last-written keys and skips the superseded ones. Restorations are expressed in the effect status (a fully clean compensation is `Applied`; a mixed one is `AppliedWithRepair` carrying only the tombstone repairs) — no new repair vocabulary.

Strand-blocks (ratified Pass 12, closing P12-K10). A `StrictInverse` undo that refuses to tombstone a minted object still referenced by a live non-member (e.g., a staff whose instance survives outside the transaction) records `ConflictKind::TransactionConflict` — the reuse is blessed: the strand-block is a transaction-scoped conflict of the undo, and the conflict record’s affected objects and description carry the strand detail. No dedicated undo conflict kind is added.

Idempotence asymmetry (ratified Pass 12, closing P12-K11). An undo’s value restorations enter the write chains as ordinary writes by the undo operation — no distinguished undo provenance.

Consequently a *second* undo of the same transaction finds each restored key superseded by the first undo and refuses (Conflicted under `StrictInverse`, skipped under `BestEffort`), while *absence* restorations (not representable as chain writes) repeat idempotently. The asymmetry is normative, documented behavior; a chain-native undo provenance would be revisited only under the deferred undo-as-operation (streaming-consistent undo) design, which subsumes this question.

Still deferred (P11-C8, narrowed): re-introducing content tombstoned by *delete* primitives (a deterministic resurrection needs a system-derived identifier derivation the ratified closed tag set does not yet include); and *Cascade*'s dependent-closure computation — *Cascade* remains `StrictInverse` over the same set. *Transpose* inversion is no longer listed: `TransposeInterval` is undone by value restoration, not by applying an inverse interval (Push 4a, closing P12-K2). An inverse $(-d, -c)$ does exist wherever both components are negatable — the reduction never saturates — but `i32::MIN` has none, so the inverse is partial and undo does not depend on it. The frozen *Transpose* is undone by neither.

vo → v1 Payload Migration

A vo envelope carries an identifier-only payload; a v1 envelope carries the value-typed payload this catalog defines. The two forms do not coexist as permanent dialects (that would double the reducer surface forever); instead the catalog ships a **one-time migration** that lifts a vo envelope to v1 using the score graph as context, applied once on read. Production code carries only v1 payloads; vo envelopes survive only as a regression corpus.

REQUIREMENT 4.1

The migration `migrate_v0_envelope(v0, context: &Score)` **MUST** be **deterministic** (two implementations migrating the same vo envelope against the same context produce byte-identical v1 envelopes) and **equivalence-preserving** (a vo envelope and its v1 migration reduce to byte-identical canonical `MaterializedState`). When a value cannot be reconstructed from the vo projection plus the context, the migration **MUST** report the envelope unmigratable rather than fabricate a value, and the bundle opens read-only.

The reference implementation (`epiphany-ops::migrate`) reconstructs the `InsertEvent` event, the `DeleteEvent` compensation, the `ChangeRegionTimeModel` model, the `SetUserSystemBreak` anchor, and the event-anchored cross-cutting structures (`Tie / Slur / Beam`) self-containedly from the vo projection; a `Spanner`, anchored by `TimeAnchors` rather than event endpoints, remains unmigratable until the projection carries them. It recovers a `RespellPitch` spelling from the context (P12-K1, Section 3.3). The migration's merge gate (`epiphany-testkit::migration`) drives the inverse direction — projecting a v1 corpus to vo and migrating it back — and asserts byte-identical reduction plus a non-vacuity guard.

K₁ — Framework Slots (Phase 3)

This chapter drafted the remaining catalogue items as framework slots. The Phase-2 **M2** expansion (the broad-Ko groups in *epiphany-ops*) implemented four groups of them; with the **M2e** catalogue expansion their full per-primitive schemas now appear in Chapter 3, so they are *normative under the Phase-2 profile* and an implementation **MUST** *not* reject them. They are cross-referenced first. The genuinely Phase-3 slots that remain **unavailable** are listed second: an implementation **MUST** reject an operation of one of *those* kinds. Each remaining slot is a schema-fill against the template of Chapter 2; adding one is not a fresh design.

Implemented since M2 (now in Chapter 3)

Modify event; identified-pitch operations; transpose M2 Group 1 — Sections 3.4, 3.5, and 3.6.

Delete / modify cross-cutting M2 Group 2 — Sections 3.9 and 3.10 (creation is Section 3.8).

Create / delete region / staff instance / voice M2 Group 3 — Section 3.12 (set-union creation and the empty-only delete).

Set metadata / metric grid / user page break M2 Group 4 — Section 3.16 (advisory metadata, structural metric grid, advisory page break).

Implemented in the Phase-3 first tranche (now in Chapter 3)

Create staff; set time signature / tempo segment; set layout Sections 3.13, 3.17, and 3.18. The disciplines are as drafted: set-union creation, LWW structural overwrite, and LWW advisory respectively.

Added in the schema-major-2 revision (Chapter 3)

Create / delete repeat structure Section 3.14 — net-new primitives (never drafted as framework slots). Set-union creation with the all-anchors-live precondition and a delete-wins tombstone; the create is born at v2, the delete stamps major 0.

Retired slots (ratified Pass 12: outside the operation set)

Create score / canvas — retired, closing P12-K8 The document root and the canvas are *structural givens*, not operation products: `TypedObjectId` has no `Canvas` kind, the root is never op-minted, and genesis is normatively the empty-document constructor plus bundle creation, outside the operation set (core specification, Chapter 5 THE CANVAS). These are not “unavailable slots” awaiting a design — no operation kind will be assigned to them. The decision is revisited only if an addressable multi-canvas model is adopted at a future schema major.

Non-Goal

The full 60–80-primitive catalogue is not a Phase-2 deliverable. The framework (Chapter 2) and the Ko representative set (Chapter 3) are sufficient to exercise every reduction discipline; the remaining primitives are Phase-3 schema-fill.